Data Definitions and Conditionals

Robert “Corky” Cartwright
Department of Computer Science
Rice University

Today's Goals

- Simple data definitions
- Template for processing simple struct data
- Inductive (self-referential) data definitions
- Conditionals
- Template for processing inductive (recursive) data

Simple Struct Data Definitions

- How do we define new forms of data in Racket? For example, say we want to write a program for the registrar that maintains a directory of courses that can be searched …
- Problem description
  - "… Each university course will have an associated department and course number, as well as a class size. …"
- Data definition

```
;; A course is a structure (make-course dept num size)
;; where dept is a symbol, and num and size are numbers
(define-struct course (dept num size))
```

- Scheme processes this *struct* definition by creating the following operations:
  - *constructor:* `make-course`,
  - *accessors:* `course-dept`, `course-num`, `course-size`
  - *recognizer:* `course?`

Creating and Using Structures

- Syntax for creating a structure:

  ```
  (define this-class (make-course 'COMP 311 41))
  ```

- A structure instance (a constructor applied to values) is a value (and hence is *not* reducible)
  - It is a constructed object. It is a value just like **`1, true, or 'Rabbit`**
  - It is a constructed object. It is NOT a reducible expression, like **`(+ 1 2)`**
- Syntax for extracting fields

  -
    ```
    (course-dept this-class)
    (course-num this-class)
    ```
- Reduction for field access

  ```
  (course-dept (make-course 'COMP 210 50)) => 'COMP
  ```

- Notes:
  - `(make-course 'COMP 210 50)` is a value
  - `(make-course 'COMP 210 size)` is *not* a value (why not?)
  - `(make-course 'COMP 210 (+ 25 25))` is *not* a value (why not?)

The Design Recipe (Again!)

How should I go about writing programs (functions)?

1. Analyze problem and define any requisite data types.
2. State the *type contract* and *purpose* for the *function* that solves the problem.
3. Give input-output examples for the function using `check-expect`.
4. Select and instantiate a data processing template for the function body that performs the intended processing.
5. Write the definition of the function by filling in the template.
6. Test it using examples (from 3.), and confirm that the tests succeeded.

The order of the steps of the recipe is important.

Template for Processing a Struct Data Type

- We start from the data definition. Example:

```
;; A course is a structure (make-course dept num size)
;; where dept is a symbol, and num and size are numbers
(define-struct course (dept num size))
```

- Template for <u>any</u> function <u>processing</u> an argument of type **`course`**

```
;; (define (f c)
;;      ... (course-dept c) ...
;;      ... (course-num c) ...
;;      ... (course-size c) ...)
```

- Examples of such a function

```
;; big-class? : course -> bool
;; empty-class? : course -> bool
;; change-dept : course dept -> course
```

Data Type → Template → Code

- Template for function processing a `course`
```
;; (define (f ... c ... )
;;   ... (course-dept c) ... (course-num c) ... (course-size c) ...)
```

- Instantiation of template for `big-class?`
```
;; (define (big-class? c)
;;   ... (course-dept c) ... (course-num c) ... (course-size c) ...)
```

- Templates help us write the code
```
   (define (big-class? c) (>= (course-size c) 30))
```

- Sophisticated types → sophisticated templates, helping us write correct, sophisticated code.

- What about types that involve multiple forms of data? Like lists? Or numbers? We need conditional operations to process them.

Conditional Expressions

- Mechanism for distinguishing different forms of input.
- Form:

```
(cond [question-1  result-1]
      [question-2  result-2]
        ...
      [question-n  result-n]
      [else        default-result])
```

- Square brackets are used above for clarity. In Racket, they are synonymous with parentheses, but balancing brackets must match.
- The `else` "clause" is optional. If omitted and none of the questions are true, the result is a run-time error (like division by zero).

Reduction of Conditional Expressions

```
•      (cond [true         result-1] ... )
                 => result-1

•      (cond [false        result-1]
             [question-2   result-2]
                ...
             [else         default-result])
    => (cond [question-2   result-2]
                ...
             [else         default-result])

•      (cond [false        result-1]
             [else         default-result])
    => default-result

•      (cond [false        result-1])
    => ERROR: all cond predicates were false
```

If Expressions

- Simplified notation for common conditional expressions.
- Form:

```
(if question result-1 result-2)
```

abbreviates:

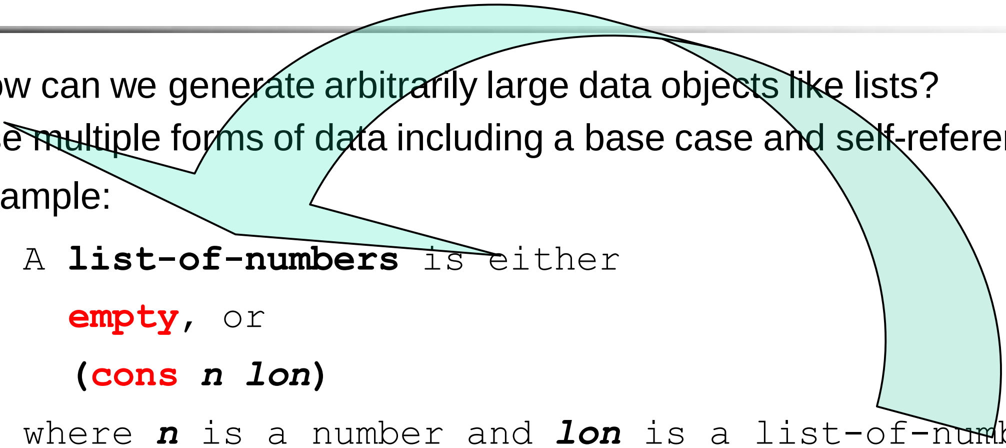
```
(cond [question   result-1]
      [else       result-2])
```

- Hence,

```
(if true result-1 result-2)   => result-1
(if false result-1 result-2) => result-2
```

Inductive Data Definitions

- How can we generate arbitrarily large data objects like lists?
- Use multiple forms of data including a base case and self-reference (induction/recursion)
- Example:



```
;; A list-of-numbers is either
;;    empty, or
;;    (cons n lon)
;; where n is a number and lon is a list-of-numbers
```

- If we assume that **`empty`** is a built-in constant identifier (like `true`), this definition can be implemented in Scheme by the `struct` definition

```
(define-struct cons (first rest))
```

- This **`struct`** definition is built-in to Racket (a primitive). For the sake of brevity, the constructor is simply called **`cons`** rather than **`make-cons`** and the accessors are called **`first`** and **`rest`** rather than **`cons-first`** and **`cons-rest`**. Note that a Racket **`struct`** definition does not stipulate the types of the fields of the structure. (An extension called Typed Racket does.) Hence, the programmer is responsible for ensuring that **`cons`** is used correctly. In teaching dialects of Racket, **`cons`** ensures that its second argument is a list. The full Racket and Scheme languages do not.

Template for Inductive Data Definition

```
;;  (define (f ... alon ...)
;;    (cond
;;      [(empty? alon ) ...]                        ;; empty case
;;      [(cons? alon ) ... (first alon) ...  ;; cons case
;;          ... (f ... (rest alon) ...) ...]))
```

- Processing inductive (self-referential) data requires recursion (self-reference) in the computation.
- Why is `cond` essential?
- In general, this form of the data definition can have more than two forms of data (any larger finite number is permitted). In addition, there is a simpler form of data definition involving multiple forms were there is no recursion; this is called a pure union type. The template for processing this form of data is identical except for absence of recursive calls.

Racket Lists

- The primary form of composite data in Racket is lists. A racket list can contain any form of data object, but in most situations, we restrict the members of a list to a specific form or type which we will call **`alpha`.**
- The following recursively defined type **`(list-of alpha)`** is built-in to Racket”

```
;; A (list-of alpha) is either:
;;    empty, or
;;    (cons n lon)
;; where n is an alpha and lon is a (list-of alpha)
```

- The constructors **`empty`**, which returns the unique empty list, and **`cons`** are built into Racket. Since there the constructors **`empty`** and **`cons`** are unique, **`(list-of alpha)`** is a subset of **`(list-of beta)`** when **`alpha`** is a subset of **`beta`**. Let **any** denote the set of Racket values. Then the general list type **`(list-of any)`** clearly contains all Racket lists and all other Racket list types are subsets of this general type.

  There is an obvious template (analogous to the template given above for **`list-of-numbers`**) for each form of built-in list **`(list-of alpha)`**.

A simple example that takes two list arguments

- The **append** function that two concatenates lists is built-in to Racket. We will show how to define this function using the name **app**

```
; app: (list-of alpha) (list-of-alpha -> list-of-alpha)
; contract: (app a b) concatenates the lists a and b.

; Examples
(check-expect (app '(a b) '(c d)) '(a b c d))
(check-expect (app empty '(c d)) '(c d))
(check-expect (app '(a b) empty) = '(a b)

; Template Instantiation (on which argument do we recur?)
|#
  (define (app x y)
    (cond [(empty? x) ...]
          [(cons? x) ... (first x) ...
             (app (rest x) y) ... ]))
#|
```

`app` cont.

-
  ```
  ; Code:
    (define (app x y)
      (cond [(empty? x) y]
            [(cons? x) (cons (first x) (app (rest x) y))]))

  ; Test?  Already done!
  ```

- Would recurring on the second argument work?

Extended Example: Insertion Sort

- Problem: given a list-of-numbers,
  sort it into ascending (non-decreasing) order.
  Look at the file _sort_.rkt in the Sample Racket Programs section of the wiki web site for the course

https://wiki.rice.edu/confluence/display/FPSCALA/Sample+Racket+Programs

Racket List Abbreviations

Every Racket list value can be laboriously expressed using the **`cons`** and **`empty`** constructors. Fortunately, Racket supports simpler notation for building lists. For any Racket expressions M_1, ..., M_n

```
(list M1 ... Mn)
```

abbreviates

```
(cons M1 (cons M2 ... (cons Mn empty) ...))
```

Example

```
(list 1 2 3)
```

abbreviates

```
(cons 1 (cons 2 ... (cons 3 empty) ...))
```

Epilog

On the homework, try to follow the design recipe in good faith. We are not particularly concerned about the exact syntax of your documenting comments. We want to see that you are faithfully following the process.